SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 9, 3–31 (1997)

Research

Porting COBOL Programs Using a Transformational Approach

JULIO CESAR SAMPAIO do PRADO LEITE,[1]* MARCELO SANT'ANNA[1] AND ANTONIO FRANCISCO do PRADO[2]

[1]*Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, R. Marquês de São Vicente 225, 22453–900, Rio de Janeiro, Brazil*
[2]*Departamento de Computação, Universidade Federal de São Carlos, Via Washington Luiz, km 235, 04499–610, São Carlos, Brazil*

SUMMARY

Transformation systems have been applied in several areas of software engineering. This paper describes the use of the transformational engine Draco-PUC in porting COBOL programs. We describe the porting strategy for going from COBOL to C/C++ and give examples of such a strategy applied to a radar application and also to a payroll system. Although targeting an object-orientated language, we do not claim to generate code that strictly follows the OO guidelines. This paper gives special attention to the knowledge structure we have been using to help guide the transformation process as well as to help in the process of design recovery.

J. Software Maintenance 9: 3–31, 1997

(No. of Figures: 30. No. of Tables: 0. No. of Refs: 20.)

KEY WORDS: software re-engineering; transformation systems; Draco paradigm; design recovery; COBOL; C++

1. INTRODUCTION

Reverse engineering is a research area of ever-increasing importance, and one in which co-operation between research work and practical work is essential. More and more, society is demanding new systems or, at least, changes in old systems. The code legacy that we have to deal with presents a challenge for software engineering. First, we have to fully understand what a legacy system does, and second, we have to change it according to new demands from our clients. Although there are already some commercially available tools to help the task of reverse engineering a system, a still bigger effort is needed to face the complexity of the existing systems, not only because of their size, but also due to their intrinsic complexity. Recent work on reverse engineering can be found in conference proceedings, such as the *Proceedings of the Working Conference on Reverse*

* Correspondence to: Julio Cesar Leite, Departamento de Informática, Pontíficia Universidade Católica do Rio de Janeiro, R. Marquês de São Vincente 225, 22453-900, Rio de Janeiro, Brazil

CCC 1040–550X/97/010003–29$17.50

Received 12 April 1996
Revised 8 October 1996

Engineering ('WCRE') since 1993, and periodicals, such as the May 1994 issue of the *Communications of the ACM*.

We view reverse engineering as an important support to re-engineering, and believe that is how maintenance should be treated. In our research (Leite, Prado and Sant'Anna, 1992) we have been using a general paradigm schema where the re-engineering process is divided into four phases called *recover*, *specify*, *redesign* and *reimplement*. Our process is geared towards the idea of reusability, reusing as much as possible during the process of re-engineering.

Porting occurs in the context of adaptive maintenance, that is, the functionality stays the same, but there is a need to alter the supporting platform (hardware or software). In such cases, most of the work is done in what we call the reimplementation phase. Using the Draco paradigm (Neighbors, 1984), most of the re-implementation is done by transformations that do not need application domain knowledge. Once we have the knowledge from the source language and the target language, we are able to specify the mapping between them and to apply this mapping to any program written in the source language.

Our work started with very little regard to design recovery and has now evolved to incorporate a program knowledge base that helps in guiding the transformations as well as supporting redesign decisions. With respect to redesign, this task is specifically limited to decisions at the implementation level.

The focus of our contribution is narrowed to our specific case of interest: to implement and use complex transformations and show the flexibility of the Draco-PUC transformation system by porting COBOL programs to C/C++. Draco-PUC is currently considerably more powerful than the original Draco implemented by Neighbors and used in the mid-1980s, as exampled in Arango *et al*. (1986). Using the experience described in this paper, we have proposed and partially validated a knowledge-base structure which helps in guiding the transformational porting and also helps in design recovery for the ported applications.

This paper aims to show how to handle porting in a structured semi-automated manner and how to link transformations with recovered program abstractions. The working results we report are of our ongoing research project, but the examples we have worked on make us believe in the power and flexibility of our proposal.

Our first section of this paper is an introduction. Section 2 reviews some available transformation systems and Section 3 details the transformation system used in the Draco-PUC engine. Section 4 deals with the general strategy used for porting COBOL programs to C/C++. Section 5 uses examples to illustrate the strategy and explains in more detail the knowledge base we are proposing and Section 6 reports on two larger examples. We conclude, Section 7, by summarizing our contribution, relating it to similar work and laying down a base for our future research.

2. TRANSFORMATION SYSTEMS

Transformation systems have been proposed, for some time now, as the backbone of revolutionary software production (Balzer, Cheatham and Green, 1993). Although the promises are still to be fulfilled, there are already localized examples of success and a significant amount of experience in the area of reverse engineering.

A transformation system is a system that manipulates programs or specifications in order to change their descriptions. Transformations can be either semantics-preserving or

not. Generally a transformation system transforms a program A into program B by applying a well-defined set of transformations that should maintain the original semantics of A in B. These transformations have to be carefully written and must be proved to guarantee the semantic equivalence of the manipulation. Below we describe some existing transformation systems.

REFINE is a commercial system produced in the USA by Reasoning Systems (1992), based on research at Kestrel (Smith, Kotik and Westfold, 1985). The central point in REFINE is a library of common Lisp functions that encapsulates the essential basis of a transformation system. The actual emphasis of Reasoning is the use of REFINE as a reverse-engineering tool, and there are already packages using REFINE that support languages such as COBOL, FORTRAN, C, and Ada.

POPART is a system with a definition language for parsing and rewriting rules (Wile, 1993). Lisp is the target language and the re-rewriting rules can be integrated with it. The architecture of POPART and its integration with Lisp provide a powerful way of manipulating software descriptions.

TAMPR is also a Lisp system, but self-ported to FORTRAN (Boyle, 1989). It uses the language Poly for defining parsers and transformations. Most of the TAMPR applications have been in porting Lisp to FORTRAN, and the resulting code is reported to be very efficient. There are also reports on TAMPR work with Pascal and C.

TXL (Cordy and Carmichael, 1993) has a very elegant transformation description language and very good documentation. Its availability on the Internet has facilitated its widespread use. Several successful applications have been reported.

3. Draco-PUC

Draco-PUC (Leite, Prado and Sant'Anna, 1993; Leite, Sant'Anna and Freitas, 1994) is a software engine being developed at PUC-Rio with the objective of developing, testing and putting into practice the ideas of the Draco paradigm (Neighbors, 1984) for domain-orientated software construction. Our project started from the original Draco machine and evolved using a re-engineering approach. The core of the Draco-PUC engine is its powerful transformation engine, which serves as the basis for our porting strategy. The next section gives an introduction to Draco-PUC architecture as well as provides a very simple example of how the transformations work.

3.1. Draco-PUC transformational framework

Below we summarize some characteristics of the Draco-PUC engine that are relevant with respect to the transformational paradigm.

- A modular transformation description language in which transformations are first-class objects. This language allows the explicit specification of both local and global transformations. Local transformations are applied to short segments of a program while global transformations encompass large, distant but related, program blocks.
- A parser generator based on the traditional Lex/Yacc syntax using an LALR(1) parsing strategy with *backtracking*. The produced parsers are responsible for parsing program specifications into Dasts—Draco abstract syntax trees. Dasts are the internal representation form used by the transformation engine.

- Transformations can be described using the Dast notation as well as domain-specific syntax representations.
- Transformations have their scopes bound by domain definitions, making it possible to lay out domain-orientated strategies for transformational applications.
- The transformation engine offers an open architecture which enables the use of control points. Control points are used to drive transformation control flows and also to connect the transformation engine to external agents.
- The transformation engine has a portable C++ implementation in an architecture that incorporates experience obtained through several years of research on the Draco-PUC project.

Transformations are applied over Draco specifications, or programs, which are descriptions written in domain languages. The grammar for a domain language is specified in an extended-BNF style syntax and a parser is generated by the Draco-PUC parser generator. Transformations can map descriptions in one language into descriptions in the same language as well as change the representation language—that is, map descriptions in one language to descriptions in other languages.

Figure 1 diagrams the use of the Draco-PUC transformation engine. First of all the input program is analysed. This step generates a Dast. Using a context filter enables the selection of a region of the Dast. The selected region is submitted to the transformation engine. This selected region is called a *program locale*. The basic control, given a set of transforms, looks for a match by navigating the Dast (Draco abstract syntax tree) in a left-to-right bottom-up fashion. Once several options of transformation exist, a rule filter selects between the several options the one to be applied. After the transformation is applied the new partial Dast is again selected for transformation using the same set of candidate transformation rules, unless there is a change in the control strategy. Sets of transformations can be chained in order to accomplish a specific aimed goal. Once transformations are finished, the Dast is pretty-printed.

A transformation rule is essentially composed of a recognition pattern, called *lhs* (left-hand side) and a replacement pattern called *rhs* (right-hand side). An example of a simple transformation is shown in Figure 2.

Besides the usual *lhs* and *rhs*, one transformation in Draco-PUC may trigger events or alter the basic flow of control. This is possible by using the idea of control points with associated code (currently, only C++ code is allowed). This code has access to the pattern matcher variables as well as to the pointers that define selected program parts.

Figure 3 shows the general transformation framework. Left-to-right, the three levels of detail on transformer organization can be seen. First (Figure 3(a)), we provide a means of encapsulating a set of transformations into a block, which has four parts. The first part is for the declaration of variables to be used at the transformations control points. The second part carries procedures that have to be activated before a set of transformations is considered. The third part is the transformer itself formed by a sequence of sets of transformations. The fourth part finalizes the process.

Second (Figure 3(b)), we provide the structure for a set of transformations. Additionally to the list of transformations, each set has an initialization part, which are actions taken before the first transformation is applied, and an end part in which actions are performed after the last transformation is applied. We also have directives to describe the control strategy for that specific set of transformations. The definition of this control strategy is

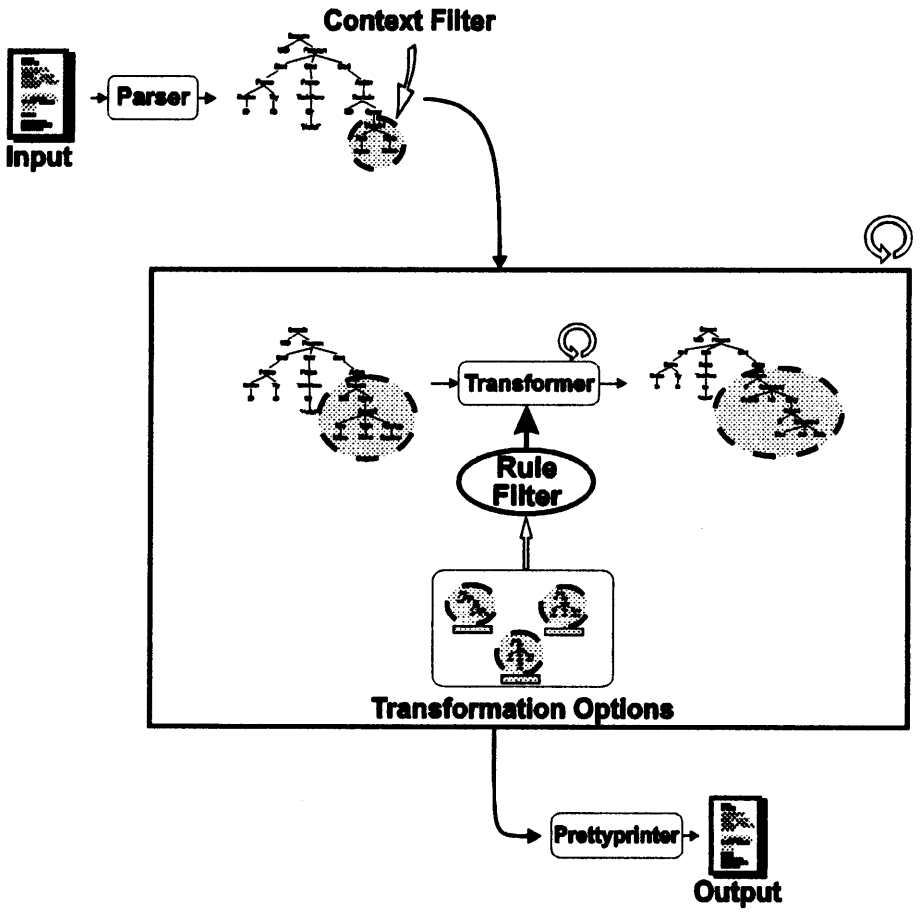


Figure 1. Draco-PUC transformation engine

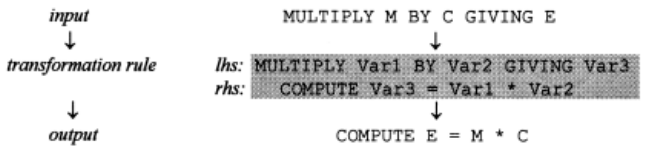


Figure 2. Example of a transformation

performed in the initialization part by choosing a triggering method, an application method and a search navigation method for the set.

Third (Figure 3(c)), we show the transformation structure itself. In addition to the *lhs* and *rhs* of a conventional production rule we have five possible control points: *pre-match*—activated each time the *lhs* is tested against the program; *match-constraint*—activated when the pattern matcher matches a variable in the *lhs* pattern to a part of the program; *post-match*—activated after a successful match between the program and the *lhs*; *pre-apply*—activated immediately before the program part is substituted by the *rhs*; *post-apply*—activated after the program has been modified.

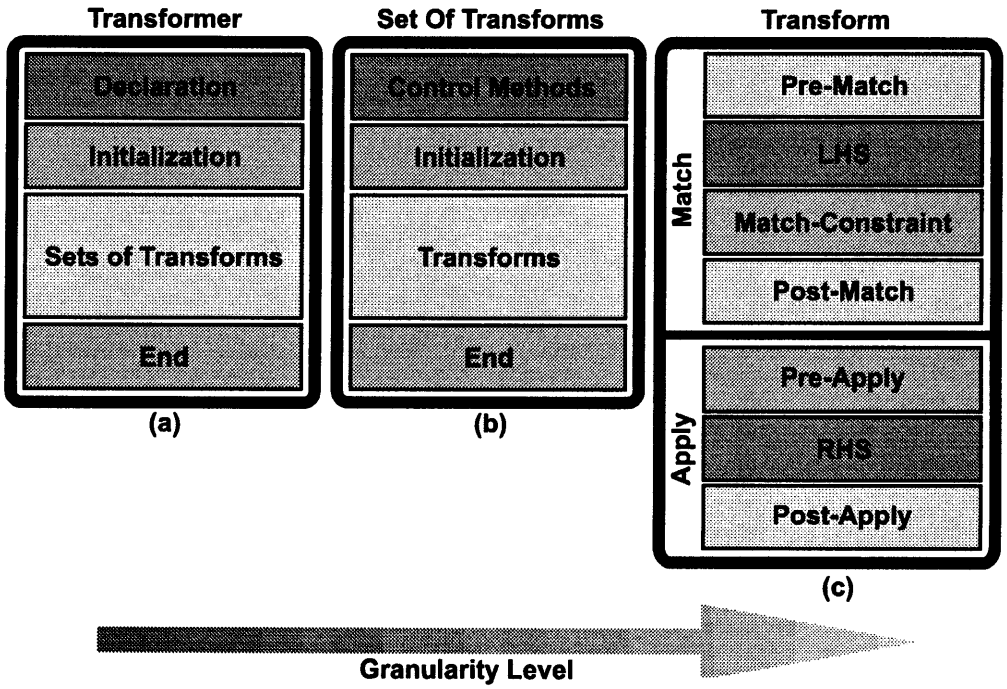


Figure 3. Transformation framework

The transformation engine uses the concept of workspace to store the Dast information it is processing. The use of workspaces makes possible the application of different policies during transformation application. For instance, if the policy is using in-place transformations, then the main workspace will hold the evolving Dast. On the other hand, if it is necessary to keep a Dast unchanged, new workspaces could be created by the transformers. Also, when it is necessary to keep a Dast configuration for some time, new workspaces could be created by the transformers.

3.2. Example

In order to provide an understanding of how Draco-PUC works, we present a classic example: symbolic differentiation implemented through transformations. This example shows how the computation of the derivative of (2+X)*(X+1) with respect to X is worked out in Draco-PUC.

Our example domain is called 'Math', encompassing natural numbers, variables and simple expressions, which handles addition, multiplication and differentiation operators. Parenthesized expressions are also allowed.

The grammar for the Math domain is presented in Figure 4. A syntactic type named exp is defined for expressions and two lexical types, NAT and VAR, are defined for naturals and variables. The lexical type IGNORE defines which characters have null value for the lexical scanner. Two left-association rules are defined for the addition and multiplication operators; their order of presentation indicates the precedence of the addition operator over the multiplication operator. In the grammar descriptions, syntactic types

```
%left '+'
%left '*'
%%
exp      : NAT
         | VAR
         | exp '+' exp
         | exp '*' exp
         | 'd' exp '/' 'd' VAR
         | '(' exp ')'
         ;
NAT      : [0-9]+
         ;
VAR      : [XY]
         ;
IGNORE   : [ \t\n]
         ;
%%
```

Figure 4. Grammar for the math domain

```
(DeriveConstant)         dC/dV = 0
(DeriveEqualVars)        dV/dV = 1
(DeriveDifferentVars)    dV1/dV2 = 0
(DeriveAddition)         d(E1+E2)/dV = dE1/dV + dE2/dV
(DeriveMultiplication)   d(E1*E2)/dV = dE1/dV * E2 + E1 * dE2/dV
(SimplifyExpPlusZero)    E+0 = E
(SimplifyZeroPlusExp)    0+E = E
(SimplifyExpMultOne)     E*1 = E
(SimplifyOneMultExp)     1*E = E

        C=constant; V,V1,V2=variables; E,E1,E2=expressions
```

Figure 5. Set of equations for symbolic differentiation

begin with a lower case character whereas lexical types begin with an upper case character. The set of grammar rules are surrounded by double per cent signs (%%). Association and precedence rules are annotated at the beginning of the grammar description using the modes %left for left association, %right for right association and %nonassoc for non-associative lexical types. Single lexemes are surrounded by single quotes (‘).

Figure 5 shows the set of equations used in the example and Figure 6 shows the

```
                    Input: d((2+X)*(X+Y))/dX

                       => d((2+X)*(X+Y))/dX
(DeriveMultiplication) => d(2+X)/dX*(X+Y) + (2+X)*d(X+Y)/dX
(DeriveAddition)       => d2/dX + dX/dX*(X+Y) + (2+X)*d(X+Y)/dX
(DeriveConstant)       => 0 + dX/dX *(X+Y)+(2+X)*d(X+Y)/dX
(DeriveEqualVars)      => 0+1 *(X+Y)+(2+X)*d(X+Y)/dX
(SimplifyZeroPlusExp)  => 1*(X+Y)+(2+X)*d(X+Y)/dX
(SimplifyOneMultExp)   => (X+Y) + (2+X)*d(X+Y)/dX
(DeriveAddition)       => (X+Y) + (2+X)*dX/dX + dY/dX
(DeriveEqualVars)      => (X+Y) + (2+X)*1 + dY/dX
(DeriveDifferentVars)  => (X+Y) + (2+X)*1 + 0
(SimplifyExpPlusZero)  => (X+Y) + (2+X)*1
(SimplifyExpMultOne)   => (X+Y)+(2+X)

                    Output: (X+Y)+(2+X)
```

Figure 6. Derivation history for d((2+*x*)*(*x*+*y*))/d*X*

derivation history for the example itself. Figure 7 lists the description of equations DeriveConstant and DeriveMultiplication using the Draco-PUC transformation language.

A transformer file has a header composed by the keyword *Transformer* followed by its name, followed by the several elements depicted in Figure 3(a). Figure 7 describes an excerpt of the transformer Symbolic.

A set of transformation rules in Draco-PUC is described by the keyword *Set Of Transforms* (see Figure 3b). In Figure 7, Derive shows two transformations from the equations presented in Figure 5. The *Control* section describes Derive as a set which is automatically tried (Trigger is automatic) when its parent transformer is invoked and which will search patterns in a bottom-up manner (Search is bottom-up), applying transformations until no more can be applied (Application is exhaustive).

A transformation rule in Draco-PUC is described by the keyword *Transform* (see Figure 3(c)). Transformations can be described using the Dast notation or using a domain specific syntax. The latter option is the most desirable. In order to use the domain syntax the transformation writer must make use of a domain escape from the Draco-PUC transformation language to the domain specific language. The domain escape is represented

```
Transformer Symbolic

  Set Of Transforms Derive

    Control
      Trigger is automatic.
      Search is bottom-up.
      Application is exhaustive.

    //
    // dC/dV = 0
    //
    Transform DeriveConstant
      Lhs: {{dast math.exp

         d [[NAT C]] / d [[VAR V]]

      }}
      Rhs: {{dast math.exp

         0

      }}

    //
    // d(E1*E2)/dV = dE1/dV*E2 + E1*dE2/dV
    //
    Transform DeriveMultiplication
      Lhs: {{dast math.exp

          d ( [[exp E1]] * [[exp E2]] )  / d [[VAR V]]

      }}
      Rhs: {{dast math.exp

          d  [[exp E1]]  / d [[VAR V]] * [[exp E2]]

                         +

          [[exp E1]] * d  [[exp E2]]  / d [[VAR V]]

      }}
```

Figure 7. DeriveConstant and DeriveMultiplication equations as Draco-PUC transformation rules

by the construct {{dast math.exp . . . }} (see Figure 7). This construct indicates that in the places where the transformation language syntax expects a Dast description, we are describing an expression in the Math domain (math.exp). Every domain syntax is extended with constructs describing pattern variables, which uses the notation: [[type varname]], where type must be a valid domain type and varname must be a symbol beginning with a letter (an identifier). For instance, in transformation DeriveConstant (Figure 7), C is a pattern variable of type NAT and V is a pattern variable of type VAR.

4. FROM COBOL TO C/C++

The general model we have developed for porting COBOL programs can be summarized by Figure 8. Our strategy has basically three steps (COBOL structuring, conversion and C/C++ structuring) and is supported by a program knowledge base. The following subsections detail each of these steps and the program knowledge base.

4.1. COBOL structuring

At this step the programs are transformed according to structured programming principles. Data flow and control flow are analysed and the program is restructured in such a way that sets of paragraphs can be grouped in procedures. The call graph between procedures is used to help clustering procedures and the data flow analysis is used to help in determining coupling measures which may lead to the definition of modules with separate compilation. When coupling is provided by memory variables, the integration of these modules is done by the linkage section. In the case of coupling via files or databases, a script language such as JCL (job control language), must be used (IBM, 1970).

The usual experience of re-engineers is the basis for this first step in our porting

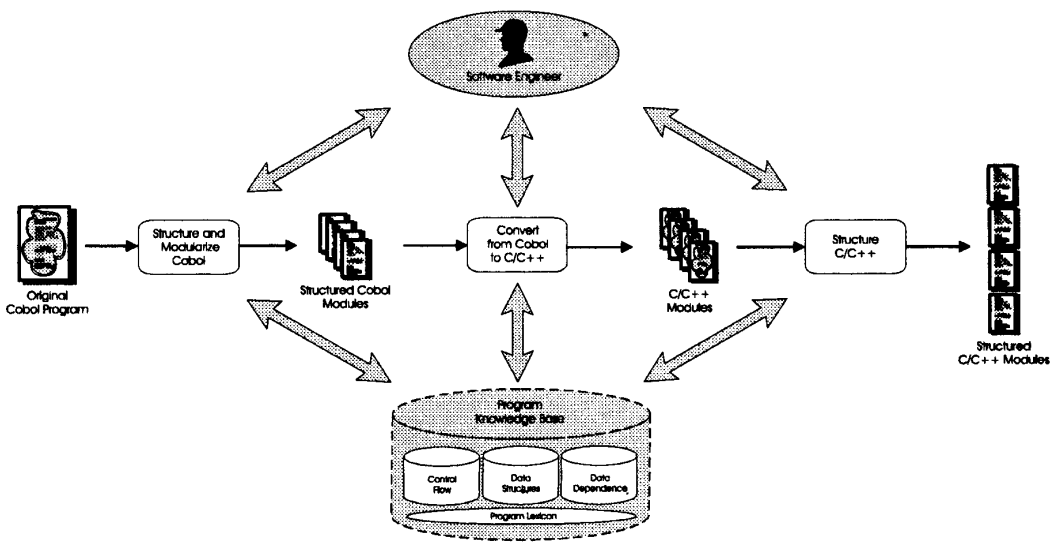


Figure 8. COBOL to C++ porting strategy

strategy: the majority of industrial COBOL programs must be first restructured prior to re-engineering (Sneed, 1995).

4.2. Conversion

The four substeps in conversion are these:

- Record descriptions are encapsulated in abstract data types in C++.
- Global variables in C++ are defined for the COBOL variables with appropriate types.
- COBOL control structures are mapped to C++ control structures.
- Paragraph blocks in COBOL are mapped to parameterless C++ functions.

After that, for each COBOL module a definition file (.h) and an implementation file with the C++ program are defined. We may also use an auxiliary C++ library which emulates some of COBOL basic types. This run-time library, named CobLib, may be linked either statically or dynamically to the generated modules. The library encapsulates the COBOL types decimal and string as well as some I/O related aspects, such as file access, screen management and report writing.

4.3. C/C++ structuring

The code resulting from the steps described above is not the usual native C/C++ code. The structuring process objective is to transform this program to make it easier to be read by C++ programmers. Using data dependence information, the global variables are distributed over functions and input and output parameters are allocated to each function. By performing these transformations we expand the structuring strategies used at the COBOL structuring step to make possible the identification of local variables, functions and abstract data types.

4.4. The program knowledge base

In order to apply our conversion strategy, information about three aspects of a program must be available: control flow, data structure and data dependence. We decided to compose a simple but functional program model which was able to encompass these three types of information. The result was the program model depicted in Figure 9. The dashed boxes in the model represent views over the data schema for programs.

The *control flow view* offers two kinds of information: intra-procedural and inter-procedural control flow information. Intra-procedural control flow information can tell which are the basic blocks in a program and how these basic blocks dominate each other. Inter-procedural control flow information stores the calling relationship between functions and procedures.

The *data dependence view* provides information on which statements declare, manipulate and use variables in the program. This very basic kind of information can assist rather complex program understanding methods such as detection of functional, object-orientated or parallel aspects of a program. Although we capture this dependence, we have not investigated further its use.

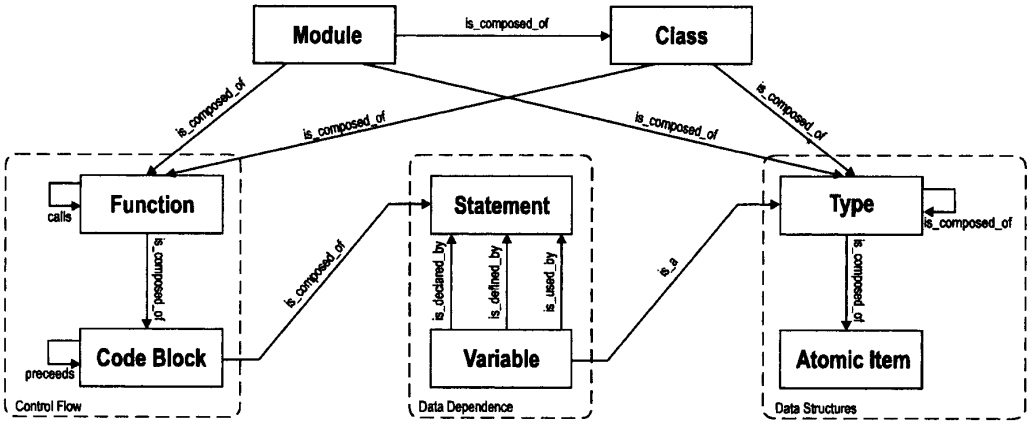


Figure 9. Program knowledge base schema

The *data structure view* provides a hierarchical decomposition of data types so that they can be understood both at a high-level and at an atomic component-parts level.

In addition to these views, we also have a *program lexicon*. The program lexicon contains all the names that characterize an instance of the meta-model (Figure 9) for a given system or program under study. Special extraction and encapsulation transformations produce the lexicon from the source code being ported. We keep each word used, and all the other words it relates to. The lexicon can be visualized in a hypertext form and should be an important link to reverse-engineering information derived from non-code sources (Leite and Cerqueira, 1995).

5. USING Draco-PUC TO IMPLEMENT THE PORTING

In order to implement a porting strategy in Draco we must go through three basic steps: 1. the construction of source and target domain parsers and pretty-printers, 2. the construction of auxiliary libraries which help in bridging the semantic gap between domains, and 3. the construction of the transformers themselves, which encapsulate the overall strategy shown in Figure 8. As also portrayed in Figure 8, besides the three basic steps of our conversion model, we need a program knowledge base in order to apply the porting transformations. Below we describe in detail all the aspects of the porting strategy. The first two steps of our porting strategy are not automated. The software engineer has to write the grammars as well as provide the auxiliary libraries.

5.1. Preparing domain parsers and pretty-printers

In order to apply our strategy we had to define the C++ and the COBOL domains. Since they are executable domains we have defined a grammar for each language, and also the pretty-printer for each language. For the COBOL grammar we have used COBOL-85 (Stern and Stern, 1990), and for C++ we have used Stroustrup's proposal (Stroustrup, 1991). The implementation of these grammars was made easier since the Draco-PUC parser generator has a backtracking mechanism, which makes possible the use of the whole class of context-free grammars.

5.2. Building auxiliary libraries

Besides the parsers and pretty-printers we also have developed a C++ run-time library named CobLib. The library encapsulates the COBOL types decimal and string as well as some I/O related aspects, such as file access, screen management and report writing. The library is used throughout the transformations as a source for classes that will be used as the basic components of the ported code. For example, the class CblString will be used in the transformations every time it is necessary to port COBOL commands that use strings. In that case the CblString will encapsulate methods that will mirror the COBOL commands.

5.3. Conversion

The automatic support for the conversion strategy is based on a three-step process. Each one defines a different set of transformations. Below we detail these steps.

5.3.1. CobStructurer

This transformer does a control flow analysis and divides the program into blocks. This division is done according to the dependence among the blocks as determined by PERFORM calls and basic block dominance. In order to do this we have annotated program parts with an auxiliary data structure. So, a set of transformations was built in order to annotate all the statements with a data structure that defines the dependencies with respect to control flow. As such, after applying the transformations we have a dependency graph for the control flow. This graph is stored as part of the program knowledge base.

The construction of the auxiliary data structure is performed by transformations without an *rhs*, in which the control point *post-match* activates the actions that fill in the data structure. After the analysis, the structuring itself takes place. The set of transformations that perform this step is based on clustering heuristics in order to form program blocks. One such heuristic is the finding of common input and output points in the flow of control.

5.3.2. CobC

The CobC transformer is composed of two main sets of transformations and several other auxiliary sets. The first one analyses the data division using the data structure view of the program knowledge base. The second set of transformations represents the porting itself, that is, it defines the semantic mappings between the structured COBOL and C++. For the definition of patterns in the transformations, we use the original domain language with pattern variables. Pattern variables are surrounded by double brackets and are typed with rules in the domain language syntax. Figure 10 shows an example of one of the CobC porting transformations.

The pattern variable name COND is typed as an expr (a COBOL expression following our COBOL syntax). The star (*) qualifier means sequence of 0 or more elements, so the pattern variable STMTS is defined as being of type sequence-of-COBOL-statements, indicating it may hold a sequence of 0 or more statements. Each pattern is enclosed by double braces, having an indication of what kind of object must be pointed by the main Dast locale pointer when trying to apply the transformation. The construction 'dast

```
/* The If1 transformation rule is a simple rewriting rule
   which finds a Cobol IF statement and then replaces it
   by a corresponding C++ IF statement.
   COND and STMTS are the names of pattern variables
   used in the transformation.
   When the match occurs, COND holds the
   conditional expression for the if statement and STMTS
   holds zero or more statements corresponding to the
   sequence of statements that are executed when the
   conditional expression is true.
*/
Transform If1
  Lhs: {{dast cobol.statement

     IF [[expr COND]] THEN [[statement* STMTS]]

  }}
  Rhs: {{dast cpp.dstatement

     if ( [[expression COND]] ) {

       [[dstatement* STMTS]]
     }
  }}
```

Figure 10. Example of CobC transformation—IF1

cobol.statement' in If1 *lhs* indicates the transformation If1 must match a COBOL statement.

Figure 11 shows another CobC transformation mapping COBOL ACCEPT statements into a C++ method invocation statement. Concerning the contents of pattern variables, COBOL INT is being mapped to C++ INTEGER_CONSTANT by the direct translation between lexical types, while the conversion of the COBOL simple_term to a C++ unary_expression is accomplished through the use of other transformations in the CobC transformer (see Figure 17). The Draco-PUC transformation engine allows direct translation between lexical types, but requires that syntactic type mappings be achieved through one or more transformations.

```
/* Transformation Accept finds Cobol ACCEPT statements and
   maps then to Accept method calls in C++. The existence
   of a method called Accept assumes CobLib is being used.
   Pattern variables X and Y hold the display position for
   the user input; they are of type integer, which means
   type INT when referring to Cobol code and
   INTEGER_CONSTANT when referring to C++ code.
   Pattern variable V holds the Cobol term containing
   the user input. V is assumed to be properly transformed
   to a C++ object which has an Accept method taking two
   integer arguments.
*/
Transform Accept
  Lhs: {{dast cobol.statement

    ACCEPT ( [[INT X]] , [[INT Y]] ) [[simple_term V]]

  }}
  Rhs: {{dast cpp.dstatement

    [[unary_expression V]].Accept( [[INTEGER_CONSTANT X]] ,
                                   [[INTEGER_CONSTANT Y]] )
  }}
```

Figure 11. Example of CobC transformation—ACCEPT

5.3.3. CStructurer

This transformer manipulates the generated C++ programs so that they can be made more readable by C++ programmers. One of the examples of this transformer is the treatment of repetition structures implemented by *if-goto-label* combinations. For these, we have written five sets of transformations.

The support transformations are parametrized by a label name and have as input a list of C++ statements. The first set of transformations verifies if there are references to the label in the list. The second set replaces the occurrences of the label by break points, assuming that the statements will be nested in a repetition structure. The other two sets transform the *if-goto-label* structures into *do-while* structures when it is possible to get rid of the labels. The last set disposes label definitions proven to be useless in the previous steps. In Figure 12 we list one of the transformations responsible for the manipulation of repetition structures.

5.4. Program knowledge base

5.4.1. Three questions

In order to define an infrastructure for a reverse-engineering effort three questions must be clearly answered:

1. *What are the relevant aspects of a program?*
2. *How must the relevant program information be extracted?*
3. *How must program information be represented and accessed?*

These questions are related to important issues on the organization of program knowledge bases. The first question addresses the need for a program model. A program model expresses the important aspects about a program from the point of view of reverse engineering. The second question addresses the methods for program information extraction. The extraction methods define how programs must be analysed so it is possible to derive the relevant information. The third question addresses the encapsulation problem. It seeks for answers on what are suitable representational methods so program information can be stored and accessed during the reverse-engineering process.

In addressing the first question, we decided to store information about control flow, data dependence and data structures. The entities and relationships that we considered relevant are described in our program knowledge base schema, as presented in Figure 9. The following two sub-sections describe how the remaining questions were treated by our team when implementing our COBOL to C++ conversion strategy.

5.4.2. Extraction procedures

We decided to use transformations to extract information used in the program model. The basic idea was to define patterns and procedures that can identify the relevant entities and relationships. For example, it is possible to collect some COBOL file declarations in the code by using the pattern shown in Figure 13. Every time this pattern successfully matches a code fragment, the variable ANAME will hold the name of a COBOL file declaration in a SELECT statement.

```
/*
   If, inside a C++ compound statement, a sequence of
   statements is followed by another sequence of
   statements starting with a label and ending in a
   condition that address this label, followed by another
   sequence of statements, then this pattern is
   candidate to define in reality a do-while statement.
   This is what precisely the lhs of transformation
   Label-If-Goto describes.
*/

Transform Label-If-Goto
  Lhs: {{dast cpp.compound_statement

    {
       [[dstatement* ST1]]

       [[IDENTIFIER LBL]]: [[dstatement ST]]
       [[dstatement* ST2]]
       if ( [[expression E]] )
         goto [[IDENTIFIER LBL]];

       [[dstatement* ST3]]
    }

  }}

  // Label-If-Goto is only applied if there is no
  // reference to label LBL inside the sequences of
  // statements ST1, ST2 and ST3.
  // This constraint is guaranteed through a test
  // performed by the application of a set of transforms
  // named FIND_LABEL_CALL. This set of transforms
  // takes as parameters a label name and a
  // sequence of statements and returns true if it
  // finds any occurrence of this label inside the
  // the sequence of statements. It returns 0 otherwise.
  Post-Match: {{dast cpp.statement_list

    if (apply("FIND_LABEL_CALL", "LBL", "ST1"))
      return(0);
    if (apply("FIND_LABEL_CALL", "LBL", "ST2"))
      return(0);
    if (apply("FIND_LABEL_CALL", "LBL", "ST3"))
      return(0);

    return(1);

  }}

  Rhs: {{dast cpp.compound_statement

    {
      [[dstatement* ST1]]

      do {
        [[dstatement ST]]
        [[dstatement* ST2]]
      } while ( [[expression E]] );

      [[dstatement* ST3]]
    }

  }}
```

Figure 12. Example of CStructurer transformation

```
Lhs: {{dast cobol.statement

          SELECT [[optional OP]] [[ID ANAME]]
                 ASSIGN TO [[file_assignment FA]]
                 [[file_attrib* FA1]]
                 FILE STATUS IS [[ID VNAME]]
                 [[file_attrib* FA2]].

}}
```

Figure 13. Finding SELECT statements

```
SELECT RADAR-FILE
    ASSIGN TO DISK
    FILE STATUS IS FSTAT
    ORGANIZATION IS LINE SEQUENTIAL.
```

Figure 14. Example of a SELECT statement

5.4.3. Encapsulation methods

We decided to use predicates on a knowledge base to store facts about programs. As an example, if we find the COBOL SELECT statement presented in Figure 14, the knowledge base should encapsulate as a minimum the facts which are shown in Figure 15.

The storage of facts when a pattern is found is performed through the use of sets of transformations we call analysers. The analysers are mainly composed of transformations having just an *lhs* and a *post-match* control point. At the control point an insertion command is issued to the inference engine communication interface which is responsible for creating the desired facts. An example of an analysis transformation is presented in Figure 16. In this example, every time a file declaration is captured, two new entries are created in the program knowledge base. One of the facts states the name of the variable responsible for mapping the physical file and the other fact defines what variable is going to hold the file status.

Similarly to the case of analysers, generators also correspond to sets of transformations in our approach. In this case, the transformations manipulate the program using the knowledge gathered by the analysers. The access to the stored facts is performed through the inference engine communication interface. The transformation shown in Figure 17 is responsible for converting to C++ a file reference occurring as a term in a COBOL expression.

In the above example, once a COBOL term is matched, inside the post-match control point, a query is constructed which asks if the variable NAME is a file-status variable. To carry on this query, the inference engine works with the program knowledge base (PKB) previously constructed by the analysers. The result of the query over the file_status predicate is stored in the variable result, which is then tested. If the term is a file-status variable, then pattern variable named CPP_NAME is set and the rhs will be instantiated. The CPP_NAME refers to the C++ corresponding name of the object that encapsulates the file being used in the program. Objects, which are associated with files, do have a

```
file(RADAR-FILE).
file_status(RADAR-FILE, FSTAT).
```

Figure 15. Example of PKB facts

```
/*
   The analysis transform Select_Extractor finds a SELECT statement
   in a Cobol program and stores in the PKB only two facts. The
   first fact defines the name of the file being declared in the
   program and the other fact describes what is the Cobol term that
   is holding the file status.
*/
Transform Select_Extractor
  Lhs: {{dast cobol.statement

           SELECT [[optional OP]] [[ID ANAME]]
                  ASSIGN TO [[file_assignment FA]]
                  [[file_attrib* FA1]]
                  FILE STATUS IS [[ID VNAME]]
                  [[file_attrib* FA2]].

  }}

  // For the storage of the facts we use the PKB method Assert and
  // the function expand.
  // The function expand takes a string and substitutes
  // the content of a pattern variable for any occurrence of a
  // pattern variable reference in the string.
  Post-Match: {{dast cpp.statement_list

    PKB.Assert(expand("is_a(file, [[ANAME]])"));
    PKB.Assert(expand("file_status([[ANAME]], [[VNAME]])"));

  }}
```

Figure 16. Example of an analysis transformation

data member named Status holding information about the file status. The rhs of File_As_An_Expression_Term retrieves the contents of Status.

6. EXAMPLES OF Draco-PUC USE

6.1. in Porting a radar cataloguing program

In order to make our strategy clearer, we show in this section portions of a porting of a radar cataloguing program (radar.cbl). Although the program is a small one (500 lines), it does have characteristic aspects of file management and user interface. It is not coded following the structured programming principles. In the next sequence of figures we present partial instances of the code at different times during the porting. Shadowed areas and dashed arrows between the figures help the reader follow the application of the conversion transformations.

Figure 18 shows fragments of the original program with a file description, some data description and some paragraphs that are related in terms of control flow. Figure 19 shows the same program fragment after the transformer CobStructurer has been applied. There is no change with respect to the data area, but at the procedure division we see the new labels: LIST-RADARS-FILE-ERR and LIST-RADARS-EXIT. These labels were introduced into the code in order to make possible, using the clustering heuristic, the creation of a block containing the paragraphs LIST-RADARS, LIST-RADARS-LOOP, LIST-RADARS-END, LIST-RADARS-FILE-ERR and LIST-RADARS-EXIT. Note that the reference to FILE-ERR is replaced by LIST-RADARS-FILE-ERR at the LIST-RADARS paragraph and that an unconditional jump to LIST-RADARS-EXIT is included before the paragraph LIST-RADARS-FILE-ERR.

```
Transform File_As_An_Expression_Term
  /*
    This transformation tests if a simple term in an expression
    is a variable which holds the status of a file.
    The test is performed through the query "file_status(*x, [[NAME]])".
    If the query is successful then the Cobol term is converted
    to the C++ Status field belonging to the associated file.
    The inference engine take as variables any identifier preceded by
    a star (*), such as the variable x in the query.
  */
  Lhs: {{dast cobol.simple_term

           [[ID NAME]]

  }}
  Post-Match: {{dast cpp.statement_list

   // Here we construct the query which tests if the content of [[ID NAME]]
   // is a file status variable and put the answer on the local variable
   // named result.
   // Variable result is of type TLEnv which means a list of environments,
   // holding all possible environment answers for the query (possible
   // bindings of x once the PKB is taken).
   TLEnv *result =
     PKB.Solve(expand("file_status(*x, [[NAME]])"));

   if (result) {
     //
     // if [[ID NAME]] is a file status variable (result==1) then the
     // pattern variable CPP_NAME is updated with the value of x, which
     // is retrieved from the first environment in result
     //
     SET("CPP_NAME", result->Retrieve("x"));
     return(1);
   }
   else
     return(0);

  }}
  Rhs: {{dast cpp.unary_expression

           [[IDENTIFIER CPP_NAME]].Status
  }}
```

Figure 17. Example of transformation using the PKB

Figures 20 and 21 show the direct porting from COBOL to C++. It is worth noting in Figure 21 the class declaration created to encapsulate the data file, and the several declarations which correspond to the COBOL declarations. It is also interesting to observe that the paragraphs grouped by the previous transformer become a function named Func_list_radars.

Figures 22 and 23 show the state of the program after the application of the transformation *Label-If-Goto* while the transformer Cstructurer is being applied. Finally, Figures 24 and 25 show the listings for the final step, when the transformer Cstructurer finishes its job. Note in the last listing (Figure 25) the label suppression and the while structure replacing the if-goto-label repetition structure.

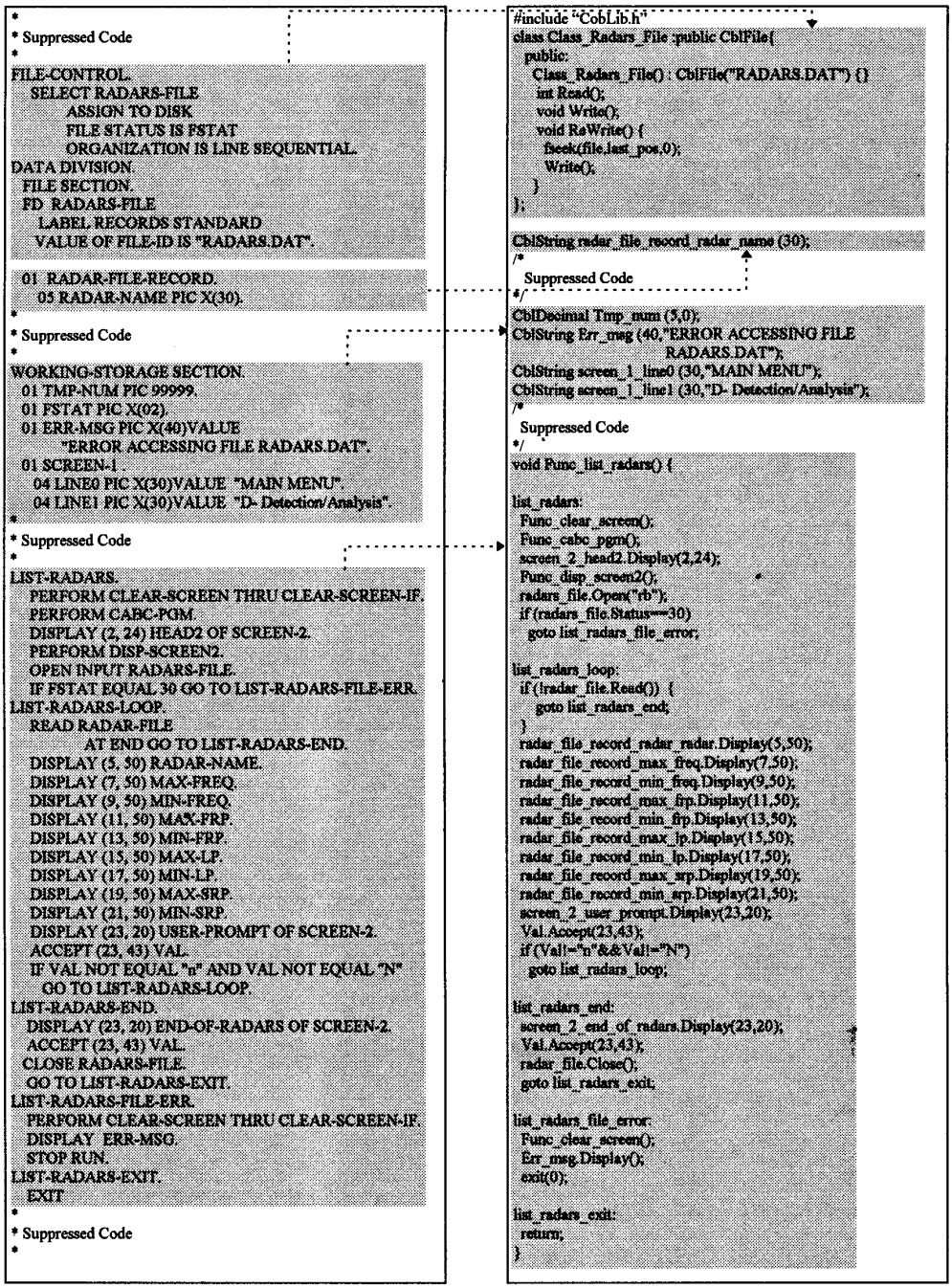
This example was run on a Sparc 2, using SunOS with 32Mb, shared on a network. It took around 36 seconds to finish the porting of the whole program which required 1112 applications of transformations from the 49 available transformations. In an Ultra 1 using Solaris, also shared in a network, the same example took one second to finish the porting.

```
*
* Suppressed Code
*
 FILE-CONTROL.
    SELECT RADARS-FILE
          ASSIGN TO DISK
          FILE STATUS IS FSTAT
          ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
  FILE SECTION.
  FD  RADARS-FILE
     LABEL RECORDS STANDARD
     VALUE OF FILE-ID IS "RADARS.DAT".
  01  RADAR-FILE-RECORD.
      05 RADAR-NAME PIC X(30).
*
* Suppressed Code
*
WORKING-STORAGE SECTION.
  01 TMP-NUM PIC 99999.
  01 FSTAT PIC X(02).
  01 ERR-MSG PIC X(40)VALUE
         "ERROR ACCESSING FILE RADARS.DAT".
  01 MAIN-SCREEN1 .
     04 LINE0 PIC X(30)VALUE  "MAIN MENU".
     04 LINE1 PIC X(30)VALUE  "D- Detection/Analysis".
*
* Suppressed Code
*
LIST-RADARS.
    PERFORM CLEAR-SCREEN THRU CLEAR-SCREEN-IF.
    PERFORM CABC-PGM.
    DISPLAY (2, 24) HEAD2 OF SCREEN-2.
    PERFORM DISP-SCRN2.
   OPEN INPUT RADARS-FILE.
    IF FSTAT EQUAL 30 GO TO FILE-ERR.
LIST-RADARS--LOOP.
    READ RADAR-FILE
       AT END,  GO TO LIST-RADARS-END.
    DISPLAY (5, 50) RADAR-NAME.
    DISPLAY (7, 50) MAX-FREQ.
    DISPLAY (9, 50) MIN-FREQ.
    DISPLAY (11, 50) MAX-FRP.
    DISPLAY (13, 50) MIN-FRP.
    DISPLAY (15, 50) MAX-LP.
    DISPLAY (17, 50) MIN-LP.
    DISPLAY (19, 50) MAX-SRP.
    DISPLAY (21, 50) MIN-SRP.
    DISPLAY (23, 20) USER-PROMPT OF SCREEN-2.
    ACCEPT (23, 43) VAL.
    IF VAL NOT EQUAL "n" AND VAL NOT EQUAL "N"
       GO TO LIST-RADARS-LOOP.
LIST-RADARS-END.
     DISPLAY (23, 20) END-OF-RADARS OF SCREEN-2.
     ACCEPT (23, 43) VAL.
     CLOSE RADARS-FILE.
*
* Suppressed Code
*
FILE-ERR.
    PERFORM CLEAR-SCREEN THRU CLEAR-SCREEN-IF.
    DISPLAY ERR-MSG.
    STOP RUN.
```

Figure 18. Original program code fragments

```
*
* Suppressed Code
*
FILE-CONTROL.
    SELECT RADARS-FILE
          ASSIGN TO DISK
          FILE STATUS IS FSTAT
          ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
  FILE SECTION.
  FD  RADARS-FILE
     LABEL RECORDS STANDARD
     VALUE OF FILE-ID IS "RADARS.DAT".
  01  RADAR-FILE-RECORD.
      05 RADAR-NAME PIC X(30).
*
* Suppressed Code
*
WORKING-STORAGE SECTION.
  01 TMP-NUM PIC 99999.
  01 FSTAT PIC X(02).
  01 ERR-MSG PIC X(40)VALUE
         "ERROR ACCESSING FILE RADARS.DAT".
  01 SCREEN-1 .
     04 LINE0 PIC X(30)VALUE  "MAIN MENU".
     04 LINE1 PIC X(30)VALUE  "D- Detection/Analysis".
*
* Suppressed Code
*
LIST-RADARS.
    PERFORM CLEAR-SCREEN THRU CLEAR-SCREEN-IF.
    PERFORM CABC-PGM.
    DISPLAY (2, 24) HEAD2 OF SCREEN-2.
    PERFORM DISP-SCREEN2.
    OPEN INPUT RADARS-FILE.
    IF FSTAT EQUAL 30 GO TO LIST-RADARS-FILE-ERR.
LIST-RADARS-LOOP.
    READ RADAR-FILE AT END GO TO LIST-RADARS-
END.
    DISPLAY (5, 50) RADAR-NAME.
    DISPLAY (7, 50) MAX-FREQ.
    DISPLAY (9, 50) MIN-FREQ.
    DISPLAY (11, 50) MAX-FRP.
    DISPLAY (13, 50) MIN-FRP.
    DISPLAY (15, 50) MAX-LP.
    DISPLAY (17, 50) MIN-LP.
    DISPLAY (19, 50) MAX-SRP.
    DISPLAY (21, 50) MIN-SRP.
    DISPLAY (23, 20) USER-PROMPT OF SCREEN-2.
    ACCEPT (23, 43) VAL.
    IF VAL NOT EQUAL "n" AND VAL NOT EQUAL "N"
       GO TO LIST-RADARS-LOOP.
LIST-RADARS-END.
    DISPLAY (23, 20) END-OF-RADARS OF SCREEN-2.
    ACCEPT (23, 43) VAL.
    CLOSE RADARS-FILE.
    GO TO LIST-RADARS-EXIT.
LIST-RADARS-FILE-ERR.
    PERFORM CLEAR-SCREEN THRU CLEAR-SCREEN-IF.
    DISPLAY ERR-MSG.
    STOP RUN.
LIST-RADARS-EXIT.
    EXIT
*
* Suppressed Code
*
```

Figure 19. Program fragments after CobStructurer



```
*
* Suppressed Code
*
FILE-CONTROL.
   SELECT RADARS-FILE
          ASSIGN TO DISK
          FILE STATUS IS FSTAT
          ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
  FILE SECTION.
  FD  RADARS-FILE
     LABEL RECORDS STANDARD
     VALUE OF FILE-ID IS "RADARS.DAT".

  01  RADAR-FILE-RECORD.
      05 RADAR-NAME PIC X(30).
*
* Suppressed Code
*
WORKING-STORAGE SECTION.
  01 TMP-NUM PIC 99999.
  01 FSTAT PIC X(02).
  01 ERR-MSG PIC X(40)VALUE
        "ERROR ACCESSING FILE RADARS.DAT".
  01 SCREEN-1 .
     04 LINE0 PIC X(30)VALUE  "MAIN MENU".
     04 LINE1 PIC X(30)VALUE  "D- Detection/Analysis".
*
* Suppressed Code
*
LIST-RADARS.
   PERFORM CLEAR-SCREEN THRU CLEAR-SCREEN-IF.
   PERFORM CABC-PGM.
   DISPLAY (2, 24) HEAD2 OF SCREEN-2.
   PERFORM DISP-SCREEN2.
   OPEN INPUT RADARS-FILE.
   IF FSTAT EQUAL 30 GO TO LIST-RADARS-FILE-ERR.
LIST-RADARS-LOOP.
   READ RADAR-FILE
            AT END GO TO LIST-RADARS-END.
   DISPLAY (5, 50) RADAR-NAME.
   DISPLAY (7, 50) MAX-FREQ.
   DISPLAY (9, 50) MIN-FREQ.
   DISPLAY (11, 50) MAX-FRP.
   DISPLAY (13, 50) MIN-FRP.
   DISPLAY (15, 50) MAX-LP.
   DISPLAY (17, 50) MIN-LP.
   DISPLAY (19, 50) MAX-SRP.
   DISPLAY (21, 50) MIN-SRP.
   DISPLAY (23, 20) USER-PROMPT OF SCREEN-2.
   ACCEPT (23, 43) VAL.
   IF VAL NOT EQUAL "n" AND VAL NOT EQUAL "N"
      GO TO LIST-RADARS-LOOP.
LIST-RADARS-END.
  DISPLAY (23, 20) END-OF-RADARS OF SCREEN-2.
  ACCEPT (23, 43) VAL.
  CLOSE RADARS-FILE.
  GO TO LIST-RADARS-EXIT.
LIST-RADARS-FILE-ERR.
   PERFORM CLEAR-SCREEN THRU CLEAR-SCREEN-IF.
   DISPLAY  ERR-MSG.
   STOP RUN.
LIST-RADARS-EXIT.
   EXIT
*
* Suppressed Code
*
```

Figure 20. Program fragments after CobStructurer

```
#include "CobLib.h"
class Class_Radars_File :public CblFile{
  public:
    Class_Radars_File() : CblFile("RADARS.DAT") {}
     int Read();
    void Write();
    void ReWrite() {
      fseek(file,last_pos,0);
      Write();
    }
};

CblString radar_file_record_radar_name (30);
/*
  Suppressed Code
*/
CblDecimal Tmp_num (5,0);
CblString Err_msg (40,"ERROR ACCESSING FILE
                       RADARS.DAT");
CblString screen_1_line0 (30,"MAIN MENU");
CblString screen_1_line1 (30,"D- Detection/Analysis");
/*
  Suppressed Code
*/
void Func_list_radars() {

list_radars:
 Func_clear_screen();
 Func_cabc_pgm();
 screen_2_head2.Display(2,24);
 Func_disp_screen2();
 radars_file.Open("rb");
 if (radars_file.Status==30)
   goto list_radars_file_error;

list_radars_loop:
 if (!radar_file.Read())  {
    goto list_radars_end;
 }
 radar_file_record_radar_radar.Display(5,50);
 radar_file_record_max_freq.Display(7,50);
 radar_file_record_min_freq.Display(9,50);
 radar_file_record_max_frp.Display(11,50);
 radar_file_record_min_frp.Display(13,50);
 radar_file_record_max_lp.Display(15,50);
 radar_file_record_min_lp.Display(17,50);
 radar_file_record_max_srp.Display(19,50);
 radar_file_record_min_srp.Display(21,50);
 screen_2_user_prompt.Display(23,20);
 Val.Accept(23,43);
 if (Val!="n"&&Val!="N")
   goto list_radars_loop;

list_radars_end:
 screen_2_end_of_radars.Display(23,20);
 Val.Accept(23,43);
 radar_file.Close();
 goto list_radars_exit;

list_radars_file_error:
 Func_clear_screen();
 Err_msg.Display();
 exit(0);

list_radars_exit:
 return;
}
```

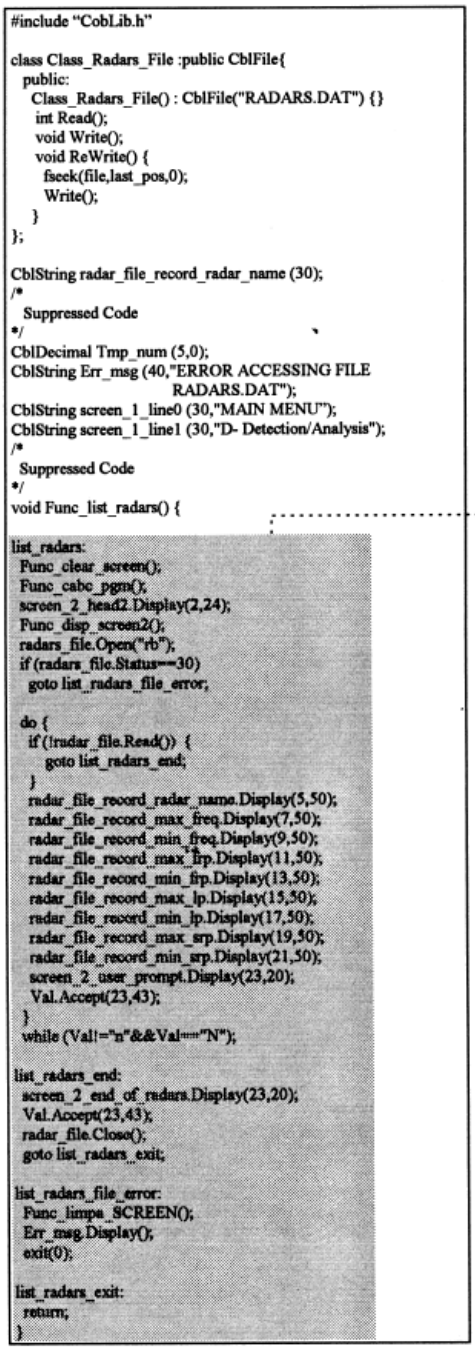
Figure 21. Program fragments after CobC

```
#include "CobLib.h"
class Class_Radars_File :public CblFile{
  public:
    Class_Radars_File() : CblFile("RADARS.DAT") {}
     int Read();
     void Write();
     void ReWrite() {
       fseek(file,last_pos,0);
       Write();
     }
};

CblString radar_file_record_radar_name (30);
/*
  Suppressed Code
*/
CblDecimal Tmp_num (5,0);
CblString Err_msg (40,"ERROR ACCESSING FILE
                       RADARS.DAT");
CblString screen_1_line0 (30,"MAIN MENU");
CblString screen_1_line1 (30,"D- Detection/Analysis");
/*
 Suppressed Code
*/
void Func_list_radars() {

list_radars:
 Func_clear_screen();
 Func_cabc_pgm();
 screen_2_head2.Display(2,24);
 Func_disp_screen2();
 radars_file.Open("rb");
 if (radars_file.Status==30)
   goto list_radars_file_error;

list_radars_loop:
 if (!radar_file.Read()) {
    goto list_radars_end;
 }
 radar_file_record_radar_radar.Display(5,50);
 radar_file_record_max_freq.Display(7,50);
 radar_file_record_min_freq.Display(9,50);
 radar_file_record_max_frp.Display(11,50);
 radar_file_record_min_frp.Display(13,50);
 radar_file_record_max_lp.Display(15,50);
 radar_file_record_min_lp.Display(17,50);
 radar_file_record_max_srp.Display(19,50);
 radar_file_record_min_srp.Display(21,50);
 screen_2_user_prompt.Display(23,20);
 Val.Accept(23,43);
 if (Val!="n"&&Val!="N")
   goto list_radars_loop;

list_radars_end:
 screen_2_end_of_radars.Display(23,20);
 Val.Accept(23,43);
 radar_file.Close();
 goto list_radars_exit;

list_radars_file_error:
 Func_clear_screen();
 Err_msg.Display();
 exit(0);

list_radars_exit:
 return;
}
```

Figure 22. Program fragments after CobC

```
#include "CobLib.h"
class Class_Radars_File :public CblFile{
  public:
    Class_Radars_File() : CblFile("RADARS.DAT") {}
     int Read();
     void Write();
     void ReWrite() {
       fseek(file,last_pos,0);
       Write();
     }
};

CblString radar_file_record_radar_name (30);
/*
  Suppressed Code
*/
CblDecimal Tmp_num (5,0);
CblString Err_msg (40,"ERROR ACCESSING FILE
RADARS.DAT");
CblString screen_1_line0 (30,"MAIN MENU");
CblString screen_1_line1 (30,"D- Detection/Analysis");
/*
 Suppressed Code
*/
void Func_list_radars() {

list_radars:
 Func_clear_screen();
 Func_cabc_pgm();
 screen_2_head2.Display(2,24);
 Func_disp_screen2();
 radars_file.Open("rb");
 if (radars_file.Status==30)
   goto list_radars_file_error;

 do {
  if (!radar_file.Read()) {
     goto list_radars_end;
  }
  radar_file_record_radar_name.Display(5,50);
  radar_file_record_max_freq.Display(7,50);
  radar_file_record_min_freq.Display(9,50);
  radar_file_record_max_frp.Display(11,50);
  radar_file_record_min_frp.Display(13,50);
  radar_file_record_max_lp.Display(15,50);
  radar_file_record_min_lp.Display(17,50);
  radar_file_record_max_srp.Display(19,50);
  radar_file_record_min_srp.Display(21,50);
  screen_2_user_prompt.Display(23,20);
  Val.Accept(23,43);
 }
while (Val!="n"&&Val=="N");

list_radars_end:
 screen_2_end_of_radars.Display(23,20);
 Val.Accept(23,43);
 radar_file.Close();
 goto list_radars_exit;

list_radars_file_error:
 Func_limpa_SCREEN();
 Err_msg.Display();
 exit(0);

list_radars_exit:
 return;
}
```

Figure 23. Use of the Label-if-Goto transformation



```
#include "CobLib.h"

class Class_Radars_File :public CblFile{
  public:
    Class_Radars_File() : CblFile("RADARS.DAT") {}
     int Read();
     void Write();
     void ReWrite() {
      fseek(file,last_pos,0);
      Write();
    }
};

CblString radar_file_record_radar_name (30);
/*
  Suppressed Code
*/
CblDecimal Tmp_num (5,0);
CblString Err_msg (40,"ERROR ACCESSING FILE
                       RADARS.DAT");
CblString screen_1_line0 (30,"MAIN MENU");
CblString screen_1_line1 (30,"D- Detection/Analysis");
/*
 Suppressed Code
*/
void Func_list_radars() {

list_radars:
 Func_clear_screen();
 Func_cabc_pgm();
 screen_2_head2.Display(2,24);
 Func_disp_screen2();
 radars_file.Open("rb");
 if (radars_file.Status==30)
  goto list_radars_file_error;

 do {
  if (!radar_file.Read()) {
     goto list_radars_end;
  }
  radar_file_record_radar_name.Display(5,50);
  radar_file_record_max_freq.Display(7,50);
  radar_file_record_min_freq.Display(9,50);
  radar_file_record_max_frp.Display(11,50);
  radar_file_record_min_frp.Display(13,50);
  radar_file_record_max_lp.Display(15,50);
  radar_file_record_min_lp.Display(17,50);
  radar_file_record_max_srp.Display(19,50);
  radar_file_record_min_srp.Display(21,50);
  screen_2_user_prompt.Display(23,20);
  Val.Accept(23,43);
 }
 while (Val!="n"&&Val!="N");

list_radars_end:
 screen_2_end_of_radars.Display(23,20);
 Val.Accept(23,43);
 radar_file.Close();
 goto list_radars_exit;

list_radars_file_error:
 Func_limpa_SCREEN();
 Err_msg.Display();
 exit(0);

list_radars_exit:
 return;
}
```

Figure 24. Program fragments during CStructurer

```
#include "CobLib.h"

class Class_radars_file :public CblFile{
  public:
    Class_radars_file() : CblFile("RADARS.DAT") {}
     int Read();
     void Write();
     void ReWrite() {
      fseek(file,last_pos,0);
      Write();
    }
};

CblString radars_file_record_radar_name (30);
/*
  Suppressed Code
*/
CblDecimal Tmp_num (5,0);
CblString Err_msg (40,"ERROR ACCESSING FILE
                       RADARS.DAT");
CblString screen_1_line0 (30,"MAIN MENU");
CblString screen_1_line1 (30,"D- Detection/Analysis");
/*
 Suppressed Code
*/
void Func_list_radars() {
 Func_clear_screen();
 Func_cabc_pgm();
 screen_2_head2.Display(2,24);
 Func_disp_screen2();
 radars_file.Open("rb");
 if (!(Arq_radar.Status==30)) {
    do {
      if (!radars_file.Read()) {
        break;
      }
      radars_file_record_radar_name.Display(5,50);
      radars_file_record_max_freq.Display(7,50);
      radars_file_record_min_freq.Display(9,50);
      radars_file_record_max_frp.Display(11,50);
      radars_file_record_min_frp.Display(13,50);
      radars_file_record_max_lp.Display(15,50);
      radars_file_record_min_lp.Display(17,50);
      radars_file_record_max_srp.Display(19,50);
      radars_file_record_min_srp.Display(21,50);
      screen_2_user_prompt.Display(23,20);
      Val.Accept(23,43);
    } while (Val!="n" && Val!="N")
    screen_2_end_of_radars.Display(23,20);
    Val.Accept(23,43);
    radars_file.Close();
    return;
 }
 Func_clear_screen();
 Err_msg.Display();
 exit(0);

}
```

Figure 25. Program fragments after CStructurer

6.2. Porting a payroll system

Another experiment was conducted using programs from the PUC-Rio payroll system. One of these programs was written in April 1985 and is still in use. It is a 7000-line program for checking the consistency of input data and printing consistency reports. These reports are generated using the report-writer feature of IBM COBOL. We will focus our example on the porting of the report section. Figure 26 shows the CobLib class CBLReport that handles the main concepts for COBOL report writer.

Given this interface, we developed transformations to fulfil this structure. A report description is captured by the *lhs* pattern depicted in Figure 27. Implemented in the Cobc transformer, three sets of transforms are chained in a hierarchical sequence to treat reports. Pattern variables' contents are handled by these specialized transformations. Figure 28 shows the apply part of a transformation used to instantiate CblReport objects based on the data extracted by the recognition pattern of Figure 27.

Figure 29 shows a fragment of the payroll system for a report description. Figure 30 shows the same program fragment in C++. The dashed arrows linking Figures 29 and 30 show how mappings between the original COBOL report description and its C++ implementation counterpart are tackled by the transformations in Cobc.

Mapping **a** shows how the parameters present in a report descriptor header become constructor parameters in a subclass of CblReport. Mapping **b** presents a line setting transformation. Mapping **c** depicts how constant strings are mapped inside a report description. Mapping **d** shows how variables are handled by the transformations. Mapping **e** displays how report totals are mapped. Mapping **f** shows the handling of repeated strings. Mapping **g** presents line increments. Mapping **h** shows how details are transformed into conditional statements inside the body of the Detail method. Following the same guidelines, control footings are mapped as is shown in Mapping **i**. Mapping **j** depicts the transformation of report footings.

The report writer part has around 300 lines of code and took five seconds on a SunOS Sparc 2 with 32 Mb of memory. The same example took less than half second to complete on an Ultra 1 using Solaris.

7. CONCLUSION

This paper describes a porting strategy based on transformations, automated through the use of the Draco-PUC transformation engine. In particular we detailed how to use a program knowledge base in conjunction with the transformational approach. Two detailed examples show how our strategy works. The novelty of our work is an extension of the transformational paradigm to use transformations to extract program structure from code, and then to use the produced information to help in the application of the porting transformations.

The Draco-PUC transformation engine is a stable working prototype which has been used in research projects at PUC-Rio as well as at the Instituto Militar de Engenharia (Military Institute of Engineering) and at Universidade Federal de São Carlos (Federal University of São Carlos, São Paulo). The C++ domain has been used in more than one project and the COBOL domain was used in the porting studies reported here. The transformers we have used and tested cover a representative set of programs, but are still far from being complete. However, we do believe that based on the work done so far, it

```
class CblReport {

  private:

    int PageLimit;
    int HeadingLine;
    int FirstDetail;
    int LastDetail;
    int FootingLine;

    int LineNumber;  //The current line number
    int PageCounter; //The current page number

    // Controls do hold the set of controls
    // used when the report is being generated
    CblReportControls Controls;

    // RH, PH, CH, DE, CF, PF and RF must be implemented inside
    // classes that have CblReport as its base class. These
    // methods encapsulate the functionallity of a report.
    void ReportHeading();
    void PageHeading();
    void ControlHeading(char* ControlName);
    void Detail(char* Detailname);
    void ControlFooting(char* ControlName);
    void PageFooting();
    void ReportFooting();

    // Show and ShowNTimes are private methods that are used
    // inside the implemenation of RH, PH, CH, DE, CF, PF and
    // RF methods.
    // Show exhibits an String at Column in
    // the current line (LineNumber).
    // ShowNTimes repeats the exhibition of String N times
    // at Column in the current line (LineNumber).
    void Show(char* String, int Column);
    void ShowNTimes(int N, char* String, int Column);

    // AssociateWith link a CblFile to a CblReport, so that
    // the output of the report be OutputFile
    void AssociateWith(CblFile* OutputFile);

  public:

    CblReport(char* control_name,
              int page_limit,
              int heading_line,
              int first_detail,
              int last_detail,
              int footing_line);

    CblReport(CblReportControls* controls,
              int page_limit,
              int heading_line,
              int first_detail,
              int last_detail,
              int footing_line);

    void Initiate();
    void Generate(char* DetailName);
    void Terminate();
};
```

Figure 26. The CblReport C++ interface

```
Lhs: {{dast cobol.report_description

        RD [[ID report_name]]
           CONTROL IS [[ID primary_control]]
           PAGE LIMIT IS [[INT page_limit]] LINES
           HEADING [[INT heading]]
           FIRST DETAIL [[INT first_detail]]
           LAST  DETAIL [[INT last_detail]]
           FOOTING [[INT footing]].

           [[report_record* records]]

}}
```

Figure 27. Recognition pattern for COBOL report descriptions

```
Pre-Apply: {{dast cpp.statement_list

        //
        // SET is a Draco API function
        // which sets a pattern variable with the contents of a
        // C++ expression.
        // CblNameToCpp takes a Cobol symbol and rewrites it as
        // a valid C++ symbol
        //
        SET("class_name", CblNameToCpp(expand("[[report_name]]")));
        //
        // As CblReport constructor initializer needs an string
        // as its first argument we surround the contents of
        // pattern variable with double quotes
        //
        SET("primary_control", expand("\"[[primary_control]]\""));
        //
        // We move the contents of workspace WSReportMethods into
        // pattern variable ReportMethodsDeclaration
        //
        MOVE("WSReportMethodsDeclaration", "ReportMethodsDeclaration");

}}
Rhs: {{dast cpp.ext_declaration.rc("CblReport.tab")

        class [[IDENTIFIER class_name]] :
           public CblReport{[[IDENTIFIER class_name]]() :
                  CblReport (  [[STRING primary_control]],
                               [[INTEGER_CONSTANT page_limit]],
                               [[INTEGER_CONSTANT heading]],
                               [[INTEGER_CONSTANT first_detail]],
                               [[INTEGER_CONSTANT last_detail]],
                               [[INTEGER_CONSTANT footing]]        ) {}

                  [[member_declaration* ReportMethodsDeclaration]]
        };
}}
```

Figure 28. Generating report descriptions in C++

is reasonable to say that our approach is very promising and worth pursuing. A research project developed at PUC-Rio (Braga, 1996) used the Draco-PUC transformation system to recover documentation information in real C++ programs. One of the largest examples of this collection of programs was a 42 module, 40 000-line program (4645 transformations from 29 applicable transformations). These were scientific programs for the research centre of Petrobrás, the Brazilian oil company.

Our strongest results are in the demonstration of the flexibility of the Draco-PUC transformation system. A special aspect of this flexibility is how the transformation system interacts with auxiliary libraries, like the CobLib and the program knowledge base.

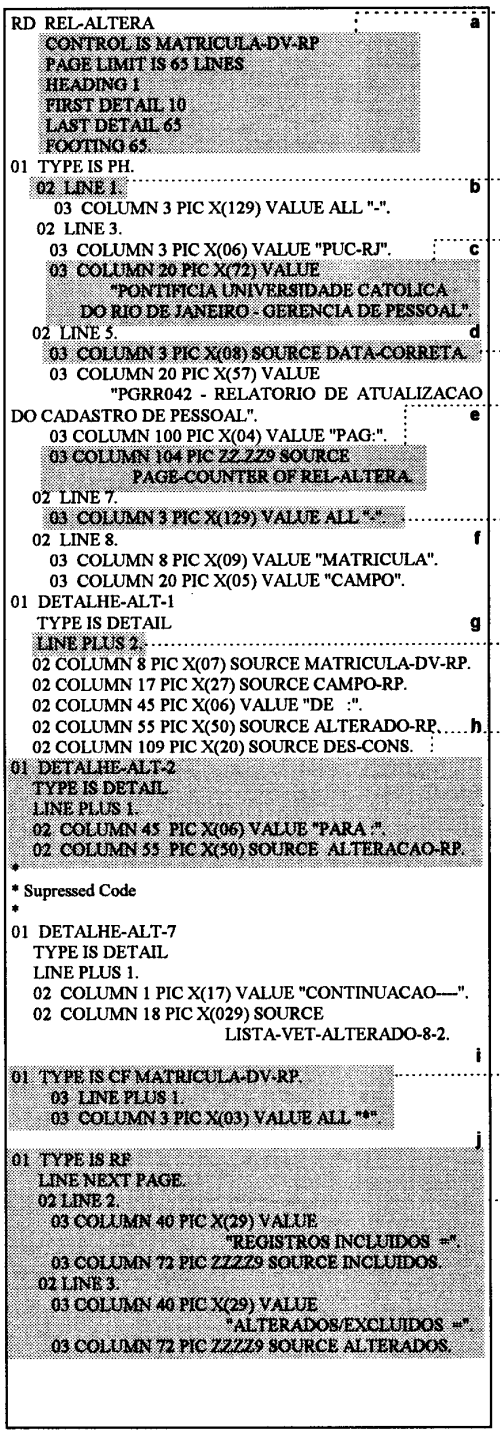


Figure 29. Example of a COBOL report description

```
class Rel_altera :public CblReport{
 Rel_altera(): CblReport("Matricula_dv_rp", 65,1,10,65,65) {}
 void Rel_altera::PageHeading();
 void Rel_altera::Detail(char * DetailName);
 void Rel_altera::ControlFooting(char * ControlName);
 void Rel_altera::ReportFooting();
};

void Rel_altera::PageHeading() {
 LineNumber=1;
 ShowNTimes(129,"-",3);
 LineNumber=3;
 Show("PUC-RJ",3);
 Show("PONTIFICIA UNIVERSIDADE CATOLICA DO RIO
DE JANEIRO - GERENCIA DE PESSOAL", 20);
 LineNumber=5;
 Show(Data_correta,3);
 Show("PGRR042 - RELATORIO DE ATUALIZACAO DO
        CADASTRO DE PESSOAL", 20);
 Show("PAG:",100);
 Show(PageCounter,104);
 LineNumber=7;
 ShowNTimes(129,"-",3);
 LineNumber=8;
 Show("MATRICULA",8);
 Show("CAMPO",20);
}

void Rel_altera::Detail(char * DetailName) {
 if (!strcmp(DetailName,"DETALHE-ALT-1")) {
   LineNumber+=2;
   Show(Matricula_dv_rp, 8);
   Show(Campo_rp,17);
   Show("DE  :",45);
   Show(Alterado_rp,55);
   Show(Des_cons,109);
 }
 if (!strcmp(DetailName,"DETALHE-ALT-2")) {
   LineNumber+=1;
   Show("PARA :",45);
   Show(Alteracao_rp,55);
 }
/*
  Supressed Code
*/
 if (!strcmp(DetailName,"DETALHE-ALT-7")) {
   LineNumber+=1;
   Show("CONTINUACAO----",1);
   Show(Lista_vet_alterado_8_2,18);
 }
}

void Rel_altera::ControlFooting(char * ControlName) {
 if (!strcmp(ControlName,"Matricula_dv_rp")) {
   LineNumber++1;
   ShowNTimes(03,"*",3);
 }
}

void Rel_altera::ReportFooting() {
 LineNumber=2;
 Show("REGISTROS INCLUIDOS =",40);
 Show(Incluidos,72);
 LineNumber=3;
 Show("ALTERADOS/EXCLUIDOS =",40);
 Show(Alterados,72);
}
```

Figure 30. Example of a report description in C++

Although the concept of a program knowledge base is not new (Jarzabek and Keam, 1995), its use in COBOL to C++ porting has been shown to be extremely useful. We also have proposed a program knowledge base (PKB) that besides holding the descriptions of the basic types, also holds descriptions of the overall structure of the program, including the control flow and data dependence. Up to now, most PKB use has been, as in Jarzabek and Keam (1995), for data structure handling, but in Draco-PUC the conversion step CobStructurer makes use of the control flow part of the PKB.

The PKB, together with the auxiliary library schema and the transformation system ability to parse C++ and COBOL constructs, are strong characteristics of our approach. Their modular representation, by the separation of concerns, makes it possible for an evolutionary approach to be deployed—that is, it is easy to incorporate new transformations as well as to write new classes in the auxiliary libraries. Of course this approach is not limited to COBOL to C++ porting, and it can be applied to other pairs of procedural languages, once we have both grammars written as Draco-PUC target domains.

Of the three steps of the conversion process, the one that had fewest transformations was the CobStructurer step. We expect that work, such as the one performed by Sneed (1995), could be used as a source for the construction of structuring transformations that will interface with the control flow part of the PKB. The strong set of transformations is the one that covers the CobC step, and which has a strong interaction with the PKB and the CobLib library. We also have observed that the many versions and dialects of COBOL require an easy to evolve parser. The radar example and the payroll example were from different versions (COBOL-74 and COBOL-85) and dialects and as such required changes to our COBOL domain.

Our experience with the COBOL to C++ porting also opened several possibilities with respect to handling reverse-engineering knowledge. Gall and Klösh (1995) proposed heuristics for finding objects based on data store entities (DSEs) and non-data store entities (NDSEs) which act primarily over tables representing basic data dependence information. The tables are called (m, u)-tables, since they store information on the manipulation and use of variables. With the help of an expert, a basic assistant model of the object-orientated application architecture is devised for the production of the final reverse generated object-orientated application model (RooAM). The introduction of transformation rules for discovering application objects as proposed by Gall and Klösh is the kind of future work which will fit well in our approach. The work of Yeh, Harris and Reubenstein (1995) proposes to find abstract data types (ADTs), using a data dependence graph between procedure and structure types as a starting point for the selection of abstract data types candidates. The procedures and structure types are the nodes of the graph, and the references by the procedures to the internal fields of the structure are the edges. We believe that our program knowledge base and the flexibility of the Draco-PUC transformation system would make it easy to incorporate such a strategy as well.

Acknowledgements

We would like to thank Prof. José Lucas Rangel for his careful work on the revision of a previous version of this article. We would also like to thank Mr Felipe Gouveia de Freitas for his ingenious solution incorporated into our parser generator, without it it would be extremely painful to handle such complex grammars as those of COBOL and C++. We also thank the Computing Facility of PUC-Rio for providing us access to some of their payroll COBOL programs.

References

Arango, G., Baxter, I. *et al*. (1986) 'TMM: software maintenance by transformation', *IEEE Software*, **3**(5), 27–39.

Balzer, R., Cheatham, T. and Green, C. (1993) 'Software technology in the 1990's: using a new paradigm', *Computer*, **16**(11), 39–45.

Boyle, J. (1989) 'Abstract programming and program transformations—an approach to reusing programs', in *Software Reusability*, Volume 1, ACM Press, New York, NY, pp. 361–413.

Braga, C. (1996) 'Uma ferramenta para geração de documentação de sistemas de software', Dissertação de Mestrado, Departamento de Informática da Pontifícia Universidade Católica do Rio de Janeiro.

Cordy, J. and Carmichael, I. (1993) 'The TXL programming language syntax and informal semantics, volume 7', Technical Report, Queen's University at Kingston, Ontario, Canada. (TXL web site is http://www.qucis.queensu.ca/STLab/TXL).

Gall, H. and Klösh, H. (1995) 'Finding objects in procedural programs: an alternative approach', in *Proceedings of the WCRE '95*, IEEE Computer Society Press, Los Alamitos, CA, pp. 208–216.

IBM (1970) *IBM System/360 Operating System: Job Control Language User's Guide*, GC28-6703, International Business Machines Corporation, Poughkeepsie, NY.

Jarzabek, S. and Keam, T. (1995). 'Design of generic reverse engineering assistant tool', in *Proceedings of the WCRE'95*, IEEE Computer Society Press, Los Alamitos, CA, pp 61–70.

Leite, J. and Cerqueira, P. (1995) 'Recovering business rules from structured analysis specifications', in *Proceedings of the WCRE'95*, IEEE Computer Society Press, Los Alamitos, CA, pp. 13–21.

Leite, J., Prado, A. and Sant'Anna, M. (1992) 'Draco-PUC, experiências e resultados de reengenharia de software', in *Proceedings of the VI Símposio Brasileiro de Engenharia de Software*, Sociedade Brasileira de Computação, Rio de Janeiro, pp. 115–128.

Leite, J., Prado, A. and Sant'Anna, M. (1993) 'Draco-PUC: a case study on software re-engineering', in *Position Paper Collection of the Second International Workshop on Software Reusability*, Memo Number 69, Software-Technologie, Universitat Dortmund, Germany, pp. 121–124.

Leite, J., Sant'Anna, M. and Freitas, F. (1994) 'Draco-PUC: a technology assembly for domain oriented software development', in *Proceedings of the Third International Conference on Software Reuse*, IEEE Computer Society Press, Los Alamitos, CA, pp. 94–101.

Neighbors, J. (1984) 'The Draco approach to constructing software from reusable components', *Transactions on Software Engineering*, **SE-10**(5), 564–574.

Reasoning Systems (1992) *REFINE User's Guide*, Reasoning Systems Inc., Palo Alto, CA.

Smith, D., Kotik, G. and Westfold, S. (1985) 'KIDS: research on knowledge-based software environments at Kestrel Institute', *Transactions on Software Engineering*, **SE-11**(11), 1278–1295.

Sneed, H. (1995) 'Reverse engineering as a bridge to CASE', in *Proceedings of the WCRE'95*, IEEE Computer Society Press, Los Alamitos, CA, pp. 300–313.

Stern, N. and Stern, R. (1990) *The Wiley COBOL Syntax Reference Guide*. John Wiley & Sons, Inc., New York, NY.

Stroustrup, B. (1991) *The C++ Programming Language*, Addison-Wesley Publishing Co., Reading, MA.

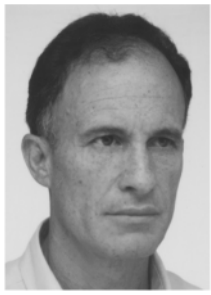
Wile, D. (1993) 'POPART: producer of parsers and related tools system builders' manual', Technical Report, USC/Information Sciences Institute, Los Angeles, CA.

Yeh, A., Harris, D. and Reubenstein, H. (1995) 'Recovering abstract data types and object instances from a conventional procedural language', in *Proceedings of the WCRE'95*, IEEE Computer Society Press, Los Alamitos, CA, pp. 227–236.

Authors' biographies:

Julio Cesar Sampaio do Prado Leite is an Associate Professor at the *Departamento de Informática* (Informatics Department) of the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) and director of the Draco-PUC project. Dr. Leite is an active researcher in the area of reuse, reverse engineering and requirements engineering, where he performed pioneer work on viewpoint analysis. He is a member of the IFIP Working Group 2.9 on software requirements engineering, and a member of an editorial board and a conference committee, as well as having served as general chair and program chair of the VII Simposio Brasileiro de Engenharia de Software in Rio in 1993. He holds a Ph.D. in Computer Science from the University of California, Irvine. E-mail: draco@inf.puc-rio.br

Marcelo Sant'Anna received a B.A. degree in Computer Engineering from PUC-Rio in 1992 and is currently a Ph.D. candidate at the Informatics Department of PUC-Rio. He has been the main designer and implementor of the Draco-PUC transformation engine since 1990. That work and his participation in several referred papers enabled him to be the first Ph.D. student admitted at PUC-Rio to the Ph.D. program without a masters degree. His main interests are transformation systems, programming language semantics and software engineering environments. E-mail: draco@inf.puc-rio.br



Antonio Francisco do Prado is an Associate Professor at the *Departamento de Computação* of the Federal University of São Carlos, São Paulo. Dr. Prado received his B.S. degrees in Military Engineering from the Military Academy of Agulhas Negras, Rio de Janeiro, in 1971, and in Mathematical Sciences from the University of Itajuba, Minas Gerais, in 1973. He received a B.S. degree in Fortification and Construction Engineering in 1979. He also received an M.S. degree in Computer Science from the Military Institute of Engineering in 1986 and a Ph.D. from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), in 1992. E-mail: draco@power.ufscar.br